Naval Research Laboratory

Washington, DC 20375-5000

NRL Memorandum Report 6718

A Paradigm for Efficient Subset Recognition

JEFFREY K. UHLMANN

Integrated Warfare Technology Branch
Information Technology Division

October 1, 1990

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	1990 October 1	

4. TITLE AND SUBTITLE

A Paradign for Efficient Subset Recognition

5. FUNDING NUMBERS

PE - 62702E
PR - 55355500
WU - DN150-127

6. AUTHOR(S)

Jeffrey K. Uhlmann

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
Washington, DC 20375-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL Memorandum Report 6718

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Project Agency
1400 Wilson Boulevard
Arlington, VA 22209-2308

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The notion of a finite-state automaton is examined with regard to problems involving the recognition of sets rather than strings. More precisely, this paper is concerned with the application of the methodology of automata theory to the general problem of efficiently determining whether a set A (or possibly a subset of A) is a subset of a set of sets S. The following contributions result from this examination:

1. The traditional notion of a finite-state automaton is shown to be impractical for use in set recognition.
2. Set-expressions, which are roughly analogous to regular expressions, are formally defined as a notation for describing member-qualified sets or classes of sets.
3. A set-recognizing automation (SRA) is formally defined and an algorithm is presented for its construction from a given set-expression.
4. More powerful forms of SRA's are developed which may have important practical applications in the area of artificial intelligence. In particular, the subset machine is developed for the fast processing of a restricted class of IF-THEN rules. Algorithm descriptions as well as LISP routines are provided.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Automata theory, Set theory IF-THEN rules, AND/OR trees	36
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIEIED	UNCLASSIFIED	UNCLASSIFIED	UL

CONTENTS

1. TRADITIONAL AUTOMATA AND SETS 1

2. SET-EXPRESSIONS 2

3. THE SET-RECOGNIZING AUTOMATION 3

4. SRA CONSTRUCTION ALGORITHM 5

5. ENHANCED CONSTRUCTION ALGORITHM 12

6. THE SUBSET MACHINE 17

7. SRA'S AND EXPERT SYSTEMS 20

8. SUMMARY 21

A PARADIGM FOR EFFICIENT SUBSET RECOGNITION

1 Traditional Automata and Sets

Consider the number of symbols in a regular expression (excluding operators and parentheses) that would be required to specify a traditional deterministic finite state automaton (DFA) capable of determining whether n input sybols form a set equal to a given set S of n elements. Since a DFA recognizes only strings, all $n!$ permutations of the n elements comprising S must be explicitly specified in the expression. Thus, a regular expression in simple sum-of-products form would require $n \cdot n!$ symbols. However, closer inspection reveals that sum-of-products form does not, in general, result in a minimal expression. Since there are only n distinct symbols, the $n!$ products may be partitioned into n groups according to their first symbol and then factored. In other words, each of the n sums of n-symbol products with identical first symbols can be converted into a product of the factored symbol and the sum of the remaining $(n-1)$-symbol products. This process may then be applied recursively on the new sums-of-products and on their resulting sums-of-products until no products within a particular partition share the same first symbol.

The number of symbols resulting after the application of this process can be described by the difference equation $f(n) = n \cdot f(n-1) + n$. This equation states that the number of symbols required to specify a machine which will recognize a set of n elements is n times the number of symbols required to specify a machine capable of recognizing a set of $(n-1)$ elements plus the n symbols which remain after factoring the common first symbols. (By definition, $f(1) = 1$ since a single symbol has no permutations other than itself.) In more numerically enlightening terms, this equation reveals that the number of symbols in a regular expression required to specify a machine capable of recognizing a set S is equal to the sum of the number of r-permutations ($r = 1$ to $|S|$) over the elements of S:

$$f(n) = \sum_{k=1}^{n} \frac{n!}{(n-k)!}.$$

From this expression it can be shown that $n! \leq f(n) \leq n!e$ and thus is $O(n!)$.

Manuscript approved January 19, 1990.

It should be clear from this result that the use of regular expressions to specify sets is impractical. However, a new form of expression shall be defined which is capable of efficiently specifying large, complex set definitions. This notation, which will be referred to as *set-expressions*, will prove to be a powerful tool for expressing a wide variety of set recognition problems.

2 Set-Expressions

Let U be a finite universal set, or domain. The sets denoted by set-expressions over U are defined recursively as follows:

1. If $\alpha \in U$, then α is a set-expression which denotes the set $\{\{\alpha\}\}$.
2. If E_1 and E_2 are set-expressions which denote sets S_1 and S_2, respectively, then $(E_1 + E_2)$ is a set-expression which denotes the set $S_1 \cup S_2$.
3. If E_1 and E_2 are set-expressions which denote sets S_1 and S_2, respectively, then $(E_1 \cdot E_2)$, optionally written E_1E_2, denotes the set $\{t_1 \cup t_2 \mid t_1 \in S_1 \text{ and } t_2 \in S_2\}$.
4. If E_1 and E_2 are set-expressions which denote sets S_1 and S_2, repectively, then $(E_1 - E_2)$ denotes the set $S_1 - S_2$ (where '-' in this case represents ordinary set difference).
5. If E_1 and E_2 are set-expressions which denote sets S_1 and S_2, repectively, then (E_1/E_2) denotes the set $\{t_1 - (t_1 \cap t_2) \mid t_1 \in S_1 \text{ and } t_2 \in S_2\}$.

(Note 1: The '·' and '+' are read as AND and OR, respectively. Note 2: Many parentheses may be avoided without confusion if the AND operator is assumed to take precedence over OR. Note 3: the '/' and '*' operators and the '-' and '+' operators are not in general inverses; e.g., $(A * B)/B = A$ if and only if every elemental set in A is disjoint with every elemental set in B. Similarly, $(A + B) - B = A$ if and only if A and B are disjoint. The '/' and '-' operators are discussed briefly in the appendix.)

For example, the set-expression ab denotes the set $\{\{a,b\}\}$. The expression $a+b$, on the other hand, specifies the set $\{\{a\},\{b\}\}$. The more complicated expression $(a+b)(c+d)$ denotes the set $\{\{a,c\}, \{a,d\}, \{b,c\}, \{b,d\}\}$. From the definition and examples, one should notice that although set-expressions operate on sets of sets, they are very similar to ordinary set notation both in terms of intuition and flexibility. In order to further parallel this similarity, the universal set U will be regarded as a set-expression denoting a set of single-element sets. This will permit expressions such as $U - a_k$ without any confusion as to what is denoted. An exponential operator can also be assumed which specifies repeated applications of the AND operation. For example, the expression $(a_1 + a_2 + ... + a_n)^k$, or just U^k, specifies the set of subsets over U, $|U| = n$, of k or fewer elements. Similarly, one can concisely define sets such as the set of all subsets over a given U minus an element a_k, or the set of all subsets over a given U all of which contain an element a_k.

3 The Set-Recognizing Automaton

An SRA (Set Recognizing Automaton), like automata for strings, consists of a finite set of states and a set of transitions which determines the subsequent state of the machine from the current state of the machine and an input symbol. It has an initial state q_0 and a set of final states F. Unlike automata for strings, SRA's are defined over a set of *elements* U, rather than an alphabet, which represents the universal set, or domain, of its inputs. This universal set differs from an alphabet in that it has an associated linear ordering θ. (The choice of an ordering function θ for any particular SRA is arbitrary simply because the ordering of the members of a set, unlike the ordering of symbols comprising a string, is completely arbitrary. For practical purposes, though, if the elements of U are viewed as strings then it is convenient to let θ represent usual lexicographic order.) All input to an SRA is assumed to be θ-ordered.

Also like automata for strings, an SRA has an associated transition diagram. This diagram

consists of a finite, directed graph in which each state is represented as a vertex and each transition from a state p to a state q on an element e_i is represented as a directed arc from the vertex p to the vertex q labelled e_i. A restriction will be imposed upon the set of allowable transitions that θ of the label of any arc leaving a particular vertex must be greater than θ of any arc entering that vertex. This restriction enforces, among other things, the definition of a set as a collection of *distinct* objects. Since an SRA assumes θ-ordered input, this restriction does not impose any practical limitations; however, its existence will be important for subsequent mathematical analyses of SRA's.

Formally, an SRA will be denoted as a 6-tuple $(Q, U, \theta, \delta, q_0, F)$ where Q is a finite set of states, U is a finite set of elements, θ is a bijection which maps the i elements in U onto the integers 1 to i, δ is a function which maps $Q \times U$ to Q (i.e. $\delta(q, e) \in Q$), q_0 is an element of Q representing the initial state of the machine, and F is a set of states from Q which represents final (or accepting) states of the machine. An SRA M is said to recognize a set $S = \{e_1, e_2, ..., e_k\}$ (ordered according to θ) if and only if e_i is a transition leaving q_0, ϵ_k is a transition to a final state, and for every element e_i, $i < k$, there exists an arc labeled ϵ_i to a state with an arc labeled e_{i+1} leaving it. In other words, $M = (Q, U, \theta, \delta, q_0, F)$ recognizes S if and only if there exists a path $e_1, e_2, ..., e_k$ from q_0 to some state $r \in F$. (Uppercase S will be used to denote a particular set recognized by a machine M while reserving the notation $S(M)$ to refer to the collection (or set) of sets which can be recognized by M.)

From the preceding definitions, some useful observations can be made concerning the maximum size (i.e. number of states) an SRA can have for a given U of n elements. Because of the imposed restriction that $\theta(e_j) > \theta(e_i)$ for every arc ϵ_j leaving a state q and every arc e_i entering q, it should be apparent that any arc e_k, $\theta(k) = k$, entering q limits the number of arcs leaving q to a maximum of $n - k$. Thus, there is a maximum of $\binom{n}{k}$ states which can be reached after consuming k symbols. Since an SRA can consume no more than n symbols, the binomial theorem implies there is a maximum of 2^n (the sum from 0 to n

of $\binom{n}{k}$) states in the entire machine. This is not surprising since as many as 2^n subsets of any set of n elements can be specified, and one would expect there to be at least one state per recognized set. (Actually, if final states are merely required to signal acceptance or non-acceptance, $2n$ final states may be coalesced into a single state, thus limiting the maximum to $2^{n-1}+1$). What is also important to note is that the linear ordering imposed by θ implies that the number of transitions in the graph is linear in the number of states.

4 SRA Construction Algorithm

In order to demonstrate the practicality of this theoretical machine, it is important to develop algorithms for implementing it in a high-level programming language. The strategy will consist of constructing a routine which will convert a given set-expression into the underlying graphical structure of its equivalent SRA and then creating a driver to animate the graph for effective set recognition. More powerful forms of SRA's will be examined and an evaluation of their capabilities and limitations will be undertaken.

The input to the construction routine will consist of a list U and a set-expression. (For expressive convenience, the use of the symbols OR and $+$ will be used interchangeably to denote the OR operation and the symbols AND and $*$ will be used interchangeably – or even omitted – to denote the AND operation.) The routine will then evaluate each operator and construct piece by piece the final machine. Each state of this machine will be represented as a triple $(a\,T\,f)$, where a is an element of U which, from the context of the machine, uniquely identifies the state; T is a list of states to which transitions can be made; and f signifies whether the state is a final state or not. Thus, at the top level the machine will appear as the transition list of the initial state. A list representation of triples is the only data structure needed. For notational convenience, the kth triple in a list L of triples will be denoted $L(k)$. Furthermore, elements of this triple will be specified by using

the notation $L(k).trans\_symb$ to access the first element of the triple, $L(k).trans\_list$ to access the second element, and $L(k).final\_state$ to refer to the third element. A function *next_triple*() will be assumed which extracts and returns the first triple from a list of triples.

The following defines a simple mechanism for evaluating a set expression:

```
function: EVALUATE (EXP);
   local variables: prod, sum, symbol;
   do forever;
      if EXP is empty then return OR(sum, prod);
      get_symb: symbol <-- NEXT_SYMBOL(EXP);
      if symbol = '*' or symbol = 'AND' then goto get_symb;
      if symbol = '+' or symbol = 'OR' then begin;
         sum <-- OR(sum, prod);
         prod <-- nil;
      end;
      else begin;
         if symbol is an expression
            then symbol <-- EVALUATE(symbol);
         else symbol <-- triple(symbol, nil, true);
         if prod is empty then prod <-- symbol;
         else prod <-- AND(symbol, prod);
      end;
   end;
end;
```

The basic logic of this function consists of a left-to-right evaluation of the expression as a sum-of-products where each symbol is an atom, an expression, or an operator. If the symbol is an atom, a triple is constructed representing a final state; if it is an expression, it

is evaluated; if it is an AND operator, it is ignored since adjacent operands are assumed to represent products; and if it is an OR operator (or the end of the expression), an OR operation is performed. Since the literals *AND*, *OR*, $*$, and $+$ are used to represent operators, they should not be elements of U. In other words, they cannot be interpreted as operands by the above algorithm.

From the *EVALUATE* function one can see that the OR operation is performed by a function *OR*. This function is defined as follows:

```
function: OR (M1, M2);
   local variables: t1, t2, machine;
   do forever;
      if M1 is empty then add M2 to machine and return machine;
      if M2 is empty then add M1 to machine and return machine;
      if M1(1).trans_symb < M2(1).trans_symb
         then add next_triple(M1) to machine;
      else if M2(1).tran_symb < M1(1).trans_symb
         then add next_triple(M2) to machine;
      else begin;
         t1 <-- next_triple(M1);
         t2 <-- next_triple(M2);
         add triple (t1.trans_symb,
                     OR(t1.trans_list, t2.trans_list),
                     t1.final_state | t2.final_state)
         to machine;
      end;
   end;
end;
```

The relatively small amount of code belies the deeper complexity of this function. Thus, a detailed explanation is necessary to demonstrate its correctness.

Observe that the OR operation in a set-expression specifies a pair of alternative membership requirements, one of which must be statisfied if a set is to be accepted by the machine described. This operation can be performed by unioning the top-level transition lists by width. *By width* is specified because it is clear that the result should have the same depth (i.e. level of nesting) as the alternative of greatest depth since the OR operation does not affect the elemental sets of the sets described by its operands. In terms of machines, this amounts to coalescing the initial states of two machines. Hence, if a trivial machine A which consists of nothing more than an empty transition list (i.e. the machine recongizes no sets) is ORed with a non-trivial machine B, the result will be a machine identical to B. Thus, the first two tests in the function merely determine whether one of the machines is trivial and, if so, returns the other unchanged.

The next three tests assume lexicographic order for θ. They determine which transition under consideration from $M1$ and $M2$ is to be placed into the union in order to maintain ascending order by θ. The transition selected is then appended to the list *machine*. If the two transitions under consideration are not distinct (i.e. represent identical states), they are coalesced into a single transition by performing an *OR* of their transition lists and a logical-or operation is necessary to ensure that the resulting state is marked as final if either of the states coalesced to produce it are final states. For example,[1]

```
M1 <-- (A (B () true C () true) false);  /* M1 = AB+AC */
M2 <-- (A (D () true) false D () true);  /* M2 = AD+D  */
OR(M1, M2);
```

[1]For notational convenience one level of parentheses will be omitted (e.g., a list of triples such as ((*a b c*) (*d e f*)) will be written as (*a b c d e f*), where triples are identified by context rather than explicitly by parentheses).

returns (*A* (*B* () *true C* () *true D* () *true*) *false D* () *true*) which is equivalent to the set-expression $A*(B+C+D)+D$, which denotes the sets AB, AC, AD, and D. An examination of the algorithm reveals that if $M1$ is a set of m sets and $M2$ is a set of n sets, then in the worst case *OR* will have to compare $m+n$ sets; however, one should expect the average case to require many fewer comparisons.

From *EVALUATE* it can be seen that the AND operation is perfomed by the function *AND*. This function is similar in logic to the *OR* function except that the unioning takes place vertically rather than horizontally. As a result, the process is applied recursively in the case of *AND* rather than iteratively as in *OR*. However, because each pair of triples from $M1 \times M2$ must be considered, the process must be repeated for every triple in $M1$ and $M2$. The definition is as follows:

```
function: AND(M1, M2);
   local variables: t1, t2, machine, temp;
   do forever;
      if M1 is empty or M2 is empty then return machine;
      if M1(1).trans_symb < M2(1).trans_symb then begin;
         t1 <-- next_triple(M1);
         t1.trans_list <-- AND(t1.trans_list, M2);
         if t1.trans_list is true then
            t1.trans_list <-- OR(t1.trans_list, M2);
         add t1 to machine;
      end;
      else if M2(1).trans_symb < M1(1).trans_symb then begin;
         t2 <-- next_triple(M2);
         t2.trans_list <-- AND(t2.trans_list, M1);
         if t2.final_state is true then
```

```
            t2.trans_list <-- OR(t2.trans_list, M1);
         add t2 to machine;
      end;
      else begin;
         t1 <-- next_triple(M1);
         t2 <-- next_triple(M2);
         fstate <-- false;
         if t2.trans_list is empty then
            fstate <-- t1.final_state;
         if t1.trans_list is empty then
            fstate <-- fstate | t2.final_state;
         temp <-- AND(t1.trans_list, t2.trans_list);
         if t1.final_state is true then
            temp <-- OR(temp, M2);
         if t2.final_state is true then
            temp <-- OR(temp, M1);
         add triple(t1.trans_symb, temp, fstate) to machine;
      end;
   end;
end;
```

A close inspection of this code reveals why each state is initially marked as final in *EVALUATE*: the process of performing a vertical union of two sets results in a filtering of final state demarcations to the transition element of greatest θ-value. In other words, *AND* ensures that a final state which signifies the acceptance of a particular set cannot be reached until all of the θ-ordered elements of that set have been consumed. Thus, states are tentatively marked as final until subsequent processing of the set-expression leads to an AND operation which explicitly specifies that it should be marked otherwise. For example,

```
M1 <-- (A (B () true C () true) false);  /* M1 = AB+AC */
M2 <-- (A (D () true) false D () true);  /* M2 = AD+D  */
AND(M1, M2);
```

returns (*A* (*B* (*D* () *true*) *false C* (*D* () *true*) *false*) *false*) which is equivalent to the set-expression $A*((B+C)*D)$, which denotes the sets ABD and ACD. An examination of the algorithm reveals that if $M1$ is a set of m sets and $M2$ is a set of n sets, then $(M1 \cdot M2)$ will cause AND to merge at most mn pairs of sets.

Given that a deterministic graphical representation of an SRA can be generated, the construction of a driver is straightforward. The following function accepts an input list S (representing a θ-ordered set over U) and an SRA M and returns a logical constant signifying whether the set is recognized by the given machine:

```
function: DRIVER (S, M);
   local variables: symbol, triple;
   if S is empty then return false;
   symbol <-- next_symb(S);
   do while M is not empty;
      triple <-- next_triple(M);
      if triple.trans_symb = symbol then begin;
         if S is empty then return triple.final_state;
         symbol <-- next_symb(S);
         M <-- triple.trans_list;
      end;
   end;
   return false;
end;
```

5 Enhanced Construction Algorithm

The machine resulting from the construction just described effectively recognizes the sets specified by its corresponding set-expression definition: however, for most practical set recognition applications, an SRA needs to reveal not only that *some* set has been recognized, but specifically *what* set has been recognized. In other words, there is a need to associate relevant information (e.g. what has been recognized, actions to be taken, etc.) about a set with the final state that accepts it. This can be accomplished through the following enhancements to the previous construction.

The following procedure accepts a list of pairs, each consisting of a set-expression and a list (or atom) of information to be associated with the final state signalling its acceptance:

```
function: DEFINITIONS (EXPRESSIONS);
   local variables: expression, machine, fs_info;
   do while EXPRESSIONS is not empty;
      expression <-- next_expression(EXPRESSIONS);
      fs_info <-- expression.final_state_information;
      machine <-- OR(machine, EVALUATE(expression.exp, fs_info));
   end;
   return machine;
end;
```

The following is an example of a definition which specifies a machine which recognizes the sets of all odd or even numbers between 1 and 4 (i.e. $U = \{1, 2, 3, 4\}$):

```
M <-- ( (1 AND 3) ODD
        (2 AND 4) EVEN );
DEFINITIONS(M);
```

The routine *DEFINITIONS* would process each pair and return the machine (1 (3 () (*ODD*)) () 2 (4 () (*EVEN*)) ()).

In order to associate the appropriate information with the appropriate final state, the *EVALUATE* routine must be enhanced so that it receives an additional argument:

```
function: EVALUATE (EXP, FS_INFO);  /* Enhanced version */
   local variables: prod, sum, symbol;
   do forever;
      if EXP is empty then return OR(sum, prod);
      get_symb: symbol <-- NEXT_SYMBOL(EXP);
      if symbol = '*' or symbol = 'AND' then goto getsymb;
      if symbol = '+' or symbol = 'OR' then begin;
         sum <-- OR(sum, prod);
         prod = nil;
      end;
      else begin;
         if symbol is an expression
            then symbol <-- EVALUATE(symbol, FS_INFO);
         else symbol <-- triple(symbol, nil, FS_INFO);
         if prod is empty then prod <-- symbol;
         else prod <-- AND(symbol, prod);
      end;
   end;
end;
```

The following changes must then be made to *OR* and *AND* to enable them to work with final state markers that are sets rather than logical constants. This entails the use of a function *union()* to handle the possibility that various set-expressions might specify non-disjoint sets

(i.e. single states can result which recognize logically different sets. These states must then have multiple information lists associated with them).

```
function: OR (M1, M2);  /* Enhanced version */
   local variables: t1, t2, machine;
   do forever;
      if M1 is empty then add M2 to machine and return machine;
      if M2 is empty then add M1 to machine and return machine;
      if M1(1).trans_symb < M2(1).trans_symb
         then add next_triple(M1) to machine;
      else if M2(1).trans_symb < M1(1).trans_symb
         then add next_triple(M2) to machine;
      else begin;
         t1 <-- next_triple(M1);
         t2 <-- next_triple(M2);
         add triple (t1.trans_symb,
                     OR(t1.trans_list, t2.trans_list),
                     union(t1.final_state, t2.final_state))
         to machine;
      end;
   end;
end;

function: AND(M1, M2);  /* Enhanced version */
   local variables: t1, t2, machine, temp;
   do forever;
      if M1 is empty or M2 is empty then return machine;
      if M1(1).trans_symb < M2(1).trans_symb then begin;
```

```
   t1 <-- next_triple(M1);
   t1.trans_list <-- AND(t1.trans_list, M2);
   if t1.trans_list is true then
      t1.trans_list <-- OR(t1.trans_list, M2);
   add t1 to machine;
end;
else if M2(1).trans_symb < M1(1).trans_symb then begin;
   t2 <-- next_triple(M2);
   t2.trans_list <-- AND(t2.trans_list, M1);
   if t2.final_state is true then
      t2.trans_list <-- OR(t2.trans_list, M1);
   add t2 to machine;
end;
else begin;
   t1 <-- next_triple(M1);
   t2 <-- next_triple(M2);
   fstate <-- false;
   if t2.trans_list is empty then
      fstate <-- t1.final_state;
   if t1.trans_list is empty then
      fstate <-- union(fstate, t2.final_state);
   temp <-- AND(t1.trans_list, t2.trans_list);
   if t1.final_state is true then
      temp <-- OR(temp, M2);
   if t2.final_state is true then
      temp <-- OR(temp, M1);
   add triple(t1.trans_symb, temp, fstate) to machine;
```

```
        end;
    end;
end;
```

The construction just described results in a machine in which each final state provides precise information about the input consumed thus far. Unfortunately, the information is probably *too precise* for many applications. For example, if the machine is driven so that information is yielded only from the state the machine happens to be in after exhausting the input, a set which does not leave the machine in a final state will merely be signalled as 'not accepted' even though a subset of that set may have led the machine through a final state. A solution to this problem might be to design the driver to output information only from the first final state reached. Such a design would guarantee that if any subsets of the input lead the machine through a final state, the machine will yield some information; however, any information about symbols consumed after that final state will be lost. Similarly, having the machine output information from only the last final state reached will not reveal whether any smaller subsets of the input were recognizable. What is needed is a machine capable of providing information about *all* subsets of the input which are recognizable.

It might appear at first that by yielding output at each final state reached during the consumption of a given input, information would be obtained about all subsets of that input which are recognizable. However, a closer inspection of the machine will reveal the inadequacy of this approach. Let M be a machine described by the set-expression $abc + b$ ($U = \{a, b, c\}$). If M is given the input ab, it will end in a non-final state even though b represents a recognizable subset. In order to enable the machine to output information about every recognizable subset for every possible input, a more sophisticated (and computationally more expensive) construction algorithm is necessary.

6 The Subset Machine

A Subset Machine is defined to be a special type of SRA denoted by a 7-tuple (Q, U, δ, q_0, Δ, λ) in which Δ is an output set representing units of information associated with particular states from Q by the relation λ. For the immediate purposes Δ will be considered to be the set of information lists associated with a set of set-expressions. However, instead of associating an element of Δ with a state if and only if that state recognizes precisely a set specified by that element's associated set-expression, λ will be considered to associate a state with an element of Δ if and only if a subset of the set recognized by that state is specified by the set-expression associated with that element. In other words, if a subset of the input consumed in reaching a given state is recognizable, then that state should recognize it. Thus, no matter what state the machine terminates in, the output from that state will include information about every recognizable subset of the consumed input.

It is clear from the above definition that the first state in a machine which is capable of recognizing a particular subset must have an information list concerning that set associated with it; however, all subsequent states must also recognize that set and, therefore, it would appear that they must also have that same information list associated with them. In order to avoid this redundant storage of information, the Subset Machine will be constructed such that only the first state which is capable of recognizing a particular set will provide information about that set. Then, if the driver is designed so that it *collects* the information available at each state during the consumption of an input, precisely the same set of information will generated upon termination of the machine as if the above definition were strictly followed. Thus, the earlier driver is enhanced as follows:

```
function: DRIVER (S, M);  /* Subset Machine */
   local variables: i, n, info;
   do while S is not empty;
```

```
      symbol <-- next_symb(S);
      n <-- number of triples in M;
      do i = 1 to n;
         if M(i).trans_symb = symbol then begin;
            M <-- M(i).trans_list;
            info <-- union(info, M(i).final_state);
            leave;  /* exit loop */
         end;
      end;
   end;
   return info;
end;
```

Essentially, all that has changed is that now an element of S for which there is no transition from the current state of M is ignored without a change of state (recall that in the earlier machines this situation implied that the input set simply could not be recognized). Also, the result now represents a *set* of information lists rather than a single one.

It is certainly not obvious from the present construction how to determine whether any subsets of the elements consumed prior to a given state are recognizable; however, this is precisely the information which must be computed for every state in the machine. Fortunately, this can be accomplished in a straightforward manner by utilizing the information already computed by the earlier construction routine. Note that every possible sequence of transitions in a machine M represents a partition of $S(M)$. For example, a transition list consisting of the k transitions $(a_1, a_2, ..., a_k)$, ordered by θ, partitions $S(M)$ such that the sets containing the elements a_j, $j \leq k$, do not contain the elements a_i where $i < j$. This fact reveals immediately that no set containing an a_i can be a subset of a set containing an a_j. Thus, to identify the recognizable subsets of a set containing an a_i, only the sets containing

an a_j where $j > i$ need to be checked. This is precisely what the following routine does in converting an SRA into a Subset Machine:

```
function: SUBSET (M);
   local variables: i, j, n;
   if M is empty then return nil;
   n <-- number of triples in M;
   do i = n to 1 by -1;
      M(i) <-- SUBSET(M(i));
      do j = 1 to (i - 1);
         M(j) <-- OR(M(i), M(j));
      end;
   end;
end;
```

An examination of the above code reveals that the conversion of an SRA to a Subset Machine is a relatively expensive process. Specifically, if it is assumed for a machine M that the elements of U are uniformly distributed throughout $S(M)$ (if the distribution of elements is not uniform, the size of the machine can be reduced by associating smaller θ-values with elements which appear more often and larger θ-values with elements that appear less often) and that the average size of the sets is roughly $|U|/2$, it is easy to see that a transition a, $\theta(a) = k$, will lead to a sub-machine of M roughly twice as large (in number of states) as a sub-machine reachable via a transition b, $\theta(b) = k + 1$. In other words, an arbitrary transition a in a machine M having n states will lead to a sub-machine of approximately $n/2^{\theta(a)}$. Thus, an SRA recognizing z sets, containing an average of $n/2$ randomly selected elements, would require $O(z\log z)$ computations for conversion to a Subset Machine.

7 SRA's and Expert Systems

Historically, expert systems grew out of attempts to create general problem solving programs using predicate logic. The idea behind expert systems is that pure logic cannot be expected to solve real-world problems without real-world knowledge. Thus, an expert system has traditionally employed both an inference engine and a knowledge base. The knowledge base usually consists of a collection of IF-THEN rules composed of logical AND and OR conditions which are evaluated by the inference engine. Although superficially similar to these IF-THEN rules, set-expressions are significantly more restrictive in that they provide no facility for variable instantiation. This restriction (among others) results in an extraordinary reduction in computational requirements while maintaining a surprising degree of expressive power. In effect, the SRA construction process can be viewed as a *compilation* of IF-THEN rules with constant operands. The result of this compilation not only processes much faster than its interpreted counterpart (i.e. inference engine and rules), but is also of a more convenient form for analyzing the logical structure of the knowledge base.

Consider a Subset Machine resulting from the construction process. Means immediately exist by which "holes" can be isolated in the knowledge base and even pinpoint ambiguous or contradictory sets of rules (where a *rule* is now considered to be a set-expression). For example, any state in the machine which recognizes two or more sets implies that the rules describing those sets fail to distinguish them for any input containing a subset equal to the set of elements consumed in reaching that state. Thus, it is a simple matter for a routine (even the construction routine) to find these ambigously-defined subsets and permit the knowledge engineer to qualify them with additional rules. In traditional expert systems these conditions are exceedingly difficult, if not impossible, to discover by means other than trial-and-error testing at the front-end of the system.

Another type of useful analysis which can be implemented is *gap analysis*. Gap analysis

consists of locating relatively long paths in the machine which do not trigger the recognition of any sets. These paths are important because they may represent holes in the knowledge base. These holes are particularly difficult to locate in traditional expert systems because if the areas of the domain are overlooked in the development stage of the knowledge base, they stand a great liklihood of being overlooked in the testing stage. Thus, gap analysis provides a means for helping to assure that the knowledge base spans its given problem domain. (Gap analysis could also be used as a primary tool for creating the knowledge base. Once the domain of expertise has been established, the knowledge engineer could successively address the largest gap in the knowledge base until the domain is fully spanned. This would provide an organized and efficient alternative to starting from scratch and "filling in the pieces.")

8 Summary

This paper began with a combinatorial argument suggesting that traditional automata theory is impractical for applications involving set recognition. A new type of automaton (a set-recognizing automaton or SRA) was then developed and a notation (set-expressions) for describing the sets accepted by such a machine. Subsequently, an algorithm for converting a set-expression to its corresponding SRA was developed. This algorithm was then enhanced to construct more powerful machines which could have applications in artificial intelligence. Specifically, the use of the Subset Machine was considered for the implementation and development of a restricted class of expert systems. These very preliminary results suggest that SRA's may have a broad range of potential applications worthy of further study.

LISP Implementation:

Here a simple LISP implementation of the algorithms described in constructing a Subset Machine is presented. The goal is to provide machine-executable code which resembles as

much as possible the algorithmic descriptions used in this paper. As a result, the LISP definitions display a heavy algol accent which is intended to make them more easily understandable to the reader who is unfamiliar with LISP.

The following routines are defined in order to provide convenient access to the various pieces of the machine:

```
(DEFUN ALPHA (L) (CAR L))
```

This function returns the first element (assumed to be an atom from the list U) of a triple.

```
(DEFUN TRANSL (L) (CAR (CDR L)))
```

This function returns the second element (assumed to be a list of triples) of a triple.

```
(DEFUN FINAL (L) (CAR (CDR (CDR L))))
```

This function returns the third element (assumed to be a list representing information about recognized sets) of the triple.

```
(DEFUN TRIPLE (L) (LIST (ALPHA L)(TRANSL L)(FINAL L))
```

This funtion returns a list whose elements constitute a triple.

```
(DEFUN REST-OF (L) (CDR (CDR (CDR L))))
```

This function takes a list L of triples and returns all but the first of those triples.

The following pages containt LISP definitions for *OR-SETS*, *AND-SETS*, *EVALUATE*, *DEFINITIONS*, *SUBSET*, and *DRIVER*. (A comparison function *LT* is assumed to exist which defines $<$ according to θ.) A simple example is then provided.

```
(DEFUN OR-SETS (M1 M2)
   (PROG (MACHINE)
   LOOP1 (COND ((NULL M1) (RETURN (APPEND MACHINE M2))))
         (COND ((NULL M2) (RETURN (APPEND MACHINE M1))))
         (COND ((LT (ALPHA M1) (ALPHA M2))
                (SETQ MACHINE (APPEND MACHINE (TRIPLE M1)))
                (SETQ M1 (REST-OF M1)))
               ((LT (ALPHA M2) (ALPHA M1))
                (SETQ MACHINE (APPEND MACHINE (TRIPLE M2)))
                (SETQ M2 (REST-OF M2)))
               (T (SETQ MACHINE
                        (APPEND MACHINE
                                (LIST (ALPHA M1)
                                      (OR-SETS (TRANSL M1)
                                               (TRANSL M2))
                                      (UNION (FINAL M1)
                                             (FINAL M2)))))
                  (SETQ M1 (REST-OF M1))
                  (SETQ M2 (REST-OF M2))))
   (GO LOOP1)))


(DEFUN AND-SETS (M1 M2)
   (PROG (MACHINE T1 T2 T1T2)
   LOOP1 (COND ((OR (NULL M1) (NULL M2)) (RETURN MACHINE)))
         (COND ((LT (ALPHA M1) (ALPHA M2))
                (SETQ T1 (LIST (ALPHA M1)
```

```
                (AND-SETS (TRANSL M1) M2)
                NIL))
        (COND ((FINAL M1)
               (SETQ T1 (LIST (ALPHA T1)
                              (OR-SETS (TRANSL T1)
                                       M2)
                              NIL))))
               (SETQ M1 (REST-OF M1))
               (SETQ MACHINE (APPEND MACHINE T1)))
        ((LT (ALPHA M2) (ALPHA M1))
         (SETQ T2 (LIST (ALPHA M2)
                        (AND-SETS (TRANSL M2) M1)
                        NIL))
         (COND ((FINAL M2)
                (SETQ T2 (LIST (ALPHA T2)
                               (OR-SETS (TRANSL T2)
                                        M1)
                               NIL))))
                (SETQ M2 (REST-OF M2))
                (SETQ MACHINE (APPEND MACHINE T2)))
        (T (SETQ T1 (AND-SETS (TRANSL M1)
                              (TRANSL M2)))
           (COND ((FINAL M1)
                  (SETQ T1 (OR-SETS
                             T1
                             (REST-OF M2)))))
           (COND ((FINAL M2)
```

```
                    (SETQ T1 (OR-SETS
                                        T1
                                        (REST-OF M1)))))
                (SETQ MACHINE
                    (APPEND MACHINE
                            (LIST (ALPHA M1)
                                  T1
                                  (UNION
                                   (COND
                                    ((TRANSL M1 NIL)
                                     (T (FINAL M1)))
                                   (COND
                                    ((TRANSL M2 NIL)
                                     (T (FINAL M2)))))))
                (SETQ M1 (REST-OF M1))
                (SETQ M2 (REST-OF M2))))
   (GO LOOP1)))


(DEFUN EVALUATE (EXP)
   (PROG (SYMBOL PROD SUM)
   LOOP (COND ((NULL EXP) (RETURN (OR-SETS SUM PROD))))
        (SETQ SYMBOL (CAR EXP))
        (SETQ EXP (CDR EXP))
        (COND ((OR (EQUAL SYMBOL 'AND) (EQUAL SYMBOL '*))
               (GO LOOP))
        (COND ((OR (EQUAL SYMBOL 'OR) (EQUAL SYMBOL '+))
```

```
                (SETQ SUM (OR-SETS SUM PROD))
                (SETQ PROD NIL)
                (GO LOOP))
               ((LISTP SYMBOL) (SETQ SYMBOL (EVALUATE SYMBOL)))
               (T (SETQ SYMBOL (LIST SYMBOL NIL FS-INFO))))
          (COND ((NULL PROD) (SETQ PROD SYMBOL))
                (T SETQ PROD (AND-SETS PROD SYMBOL))))
   (GO LOOP)))


(DEFUN DEFINITIONS (EXPRESSIONS)
   (PROG (MACHINE EXP)
   LOOP (COND ((NULL EXPRESSIONS) (RETURN (SUBSET MACHINE))))
        (SETQ EXP (CAR EXPRESSIONS))
        (SETQ EXPRESSIONS (CDR EXPRESSIONS))
        (SETQ FS-INFO (CAR EXPRESSIONS))
        (SETQ EXPRESSIONS (CDR EXPRESSIONS))
        (SETQ MACHINE (OR-SETS MACHINE (EVALUATE EXP)))
   (GO LOOP)))


(DEFUN SUBSET (M)
   (COND ((NULL M) NIL)
         (T (APPEND (LIST (ALPHA M)
                          (SUBSET (OR-SETS (TRANSL M)
                                           (REST-OF M)))
                          (FINAL M))
```

```
                    (SUBSET (REST-OF M))))))


(DEFUN DRIVER (S M)
   (PROG (INFO TEMP)
   LOOP (COND ((NULL S) (RETURN INFO)))
        (SETQ SYMBOL (CAR S))
        (SETQ S (CDR S))
        (SETQ TEMP M)
        LOOP2 (COND ((NULL TEMP) (GO LOOP))
                    ((EQUAL SYMBOL (ALPHA TEMP))
                     (SETQ M (TRANSL TEMP))
                     (SETQ INFO (UNION INFO (FINAL TEMP)))
                     (GO LOOP))
                    (T (SETQ TEMP (REST-OF TEMP))))
        (GO LOOP2)))
```

A EXAMPLE

The following defines a machine via three set-expressions:

```
(SETQ EXPRESSION '( (1 2 + 1 1) (120R13)
                    (1 2 3) (123)
                    (1 + 3) (10R3) ) )
```

EXPRESSION is then converted to a machine M:

```
(SETQ M (DEFINITIONS EXPRESSION))
```

The DRIVER can then be used to examine some set (represented by a list that is assumed to have been ordered by θ). For example,

```
(DRIVER '(1 2) M)  ==>  (120R13 10R3)

(DRIVER '(1 2 3) M)  ==>  (120R13 123 10R3)

(DRIVER '(2 3) M)  ==>  (10R3)
```

Additional Operators:

For completeness the complements of the *AND* and *OR* operations should be considered. These operations have not been discussed thus far because, although quite powerful in terms of discriptive conciseness, their indiscriminate use may result in excessive computations. Furthermore, determining what sets are described by expressions involving these operations tends to be far less intuitive than for expressions using only *AND*'s and *OR*'s; therefore, their use may be prone to error (or huge computational expense) if extreme care is not taken. Recall their definitions:

4. If E_1 and E_2 are set-expressions which denote sets S_1 and S_2, repectively, then $(E_1 - E_2)$ denotes the set $S_1 - S_2$ (where '-' in this case represents ordinary set difference).

5. If E_1 and E_2 are set-expressions which denote sets S_1 and S_2, repectively, then (E_1/E_2) denotes the set $\{t_1 - (t_1 \cap t_2) \mid t_1 \in S_1 \text{ and } t_2 \in S_2\}$.

Thus, the set-expression $a - b$ denotes the set of sets in a which are not in b. For example, the expression $U^n - U^m$ denotes the set of all sets of size ℓ, $m < \ell \leq n$, over U. The expression a/b, on the other hand, denotes the sets in a which do not have a subset in b. For example, the expression U^n/ab, $|U| = n$, denotes the set of all sets over U which do not contain the elements a and b. The algorithm for implementing these operations are as

follows:

```
function: WIDTH_DIFFERENCE (M1, M2);
   local variables: machine, t1, t2, temp, fstate;
   do forever;
      if M1 is empty then return machine;
      if M2 is empty then add M1 to machine and return machine;
      if M1.trans_symb < M2.trans_symb then
         add next_triple(M1) to machine;
      else if M2.trans_symb < M1.trans_symb then
         temp <-- next_triple(M2);
      else begin;
         t1 <-- next_triple(M1);
         t2 <-- next_triple(M2);
         if M2.final_state is not empty then fstate <-- nil;
         else fstate < M1.final_state;
         temp <-- triple(t1.trans_symb,
                         WIDTH_DIFFERENCE (t1.trans_list,
                                           t2.trans_list),
                         fstate);
         if temp.trans_list is not empty or temp.final_state
            is not empty then add temp to machine;
      end;
   end;
end;


function: DEPTH_DIFFERENCE (M1, M2);
```

```
local variables: machine, t1, t2, temp, fstate;
do forever;
   if M1 is empty or M2 is empty then return machine;
   if M2.trans_symb < m1.trans_symb then
      temp <-- next_triple(M2);
   else begin;
      t1 <-- next_triple(M1);
      if t1.trans_symb < M2.trans_symb then
         temp <-- triple(t1.trans_symb,
                         DEPTH_DIFFERENCE(t1.trans_list, M2),
                         t1.final_state);
      else begin;
         t2 <-- next_triple(M2);
         if M2.final_state is empty then
            temp <-- triple(t1.trans_symb,
                            DEPTH_DIFFERENCE(t1.trans_list,
                                             t2.trans_list),
                            t1.final_state);
         else temp <-- triple(nil, nil, nil);
      end;
      if temp.trans_list is not empty or temp.final_state
         is not empty then add temp to machine;
    end;
  end;
end;
```

These operations may be translated into LISP as follows:

```
(DEFUN WIDTH-DIFFERENCE (M1 M2)
   (PROG (MACHINE TEMP)
   LOOP (COND ((NULL M1) (RETURN MACHINE))
              ((NULL M2) (RETURN (APPEND MACHINE M1)))
              ((LT (ALPHA M2) (ALPHA M1))
               (SETQ M2 (REST-OF M2)))
              ((LT (ALPHA M1) (ALPHA M2))
               (SETQ M1 (REST-OF M1)))
              (T (SETQ TEMP
                   (LIST (ALPHA M1)
                         (WIDTH-DIFFERENCE (TRANSL M1)
                                           (TRANSL M2))
                   (COND ((FINAL M2) NIL)
                         (T (FINAL M1)))))
                 (COND ((OR (TRANSL TEMP) (FINAL TEMP))
                        (SETQ MACHINE (APPEND MACHINE TEMP))))
                 (SETQ M1 (REST-OF M1))
                 (SETQ M2 (REST-OF M2))
   (GO LOOP)))


(DEFUN DEPTH-DIFFERENCE (M1 M2)
   (PROG (MACHINE TEMP)
   LOOP1 (COND ((NULL M1) (RETURN MACHINE))
               ((NULL M2) (RETURN (APPEND MACHINE M1)))
               ((LT (ALPHA M2) (ALPHA M1))
                (SETQ M2 (REST-OF M2))
```

```
            (GO LOOP1))
           ((LT (ALPHA M1) (ALPHA M2))
            (SETQ TEMP
             (LIST (ALPHA M1)
                   (DEPTH-DIFFERENCE (TRANSL M1) M2)
                   (FINAL M1))))
           (T (SETQ TEMP
               (COND ((FINAL M2)
                      (LIST NIL NIL NIL))
                     (T (LIST (ALPHA M1)
                              (DEPTH-DIFFERENCE
                               (TRANSL M1) (TRANSL M2))
                              (FINAL M1)))))
              (SETQ M2 (REST-OF M2))
      (SETQ M1 (REST-OF M1))
      (COND ((OR (TRANSL TEMP) (FINAL TEMP))
             (SETQ MACHINE (APPEND MACHINE TEMP))))
(GO LOOP1)))
```